Distributed Backup Service

Sistemas Distribuídos

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Rúben José da Silva Torres - up201405612

João Miguel Matos Monteiro - 201506130

3 Abril de 2018

Descrição da solução para concorrência entre várias threads

Tendo em conta que criar e terminar threads tem algum "overhead" foram usados alguns *ExecutorService's* (java.util.concurrent.ExecutorService) com uma *Fixed Thread Pool*, esta classe internamente cria uma *ThreadPoolExecutor* para gerir várias threads. Foram usadas vários *ExecutorService's* para haver uma maior diversidade do tipo de threads.

Chamada de um subprotocolo por rmi

A implementação do rmi tem acesso a um *ExecutorService* com um *Fixed Thread Pool* de 3 threads. Na chamada de qualquer um dos subprotocolos por rmi são criados os objetos que implementam *runnable* referente a este mesmo subprotocol e feita a execução destes pelo *ExecutorService*, cada *Peer* pode correr apenas 3 subprotocolos ao mesmo tempo.

Subprotocolo backup

Foi considerado que o subprotocolo *backup* seria o mais utilizado, sendo assim o mesmo contém a sua própria *Fixed Thread Pool* de tamanho 8, após a divisão de um ficheiro em vário *chunks*, é executado pelo *ExecutorService* deste protocolo um *Runnable* que envia para cada um destes *chunks* a mensagem para o *multicast data channel (MDB)*, e espera pela resposta resposta dos outros *peers*. Outros subprotocolos não necessitam de uma Fixed Thread Pool própria, pois não necessitam de verificar se obtiveram respostas dos outros *peers*.

Canais Multicast

Cada canal multicast tem a sua *Fixed Thread Pool* de tamanho 5, para haver uma maior diversidade no tipo de mensagens na network dos peers. Após a receção de uma mensagem no canal o *ExecutorService* do mesmo executa um *worker* que decifra e age de acordo com a mesma. Os próprios canais multicast apesar de terem threads á escuta activa de mensagens não necessitam de estar inseridos numa *Thread Pool* pois estas só irão terminar aquando da terminação do próprio processo (peer).

Concorrência das várias threads pela “Database”

Com várias threads a aceder e alterar a mesma “Database” tivemos de ter em conta que as mesmas poderiam alterar esta “Database” exatamente ao mesmo tempo existindo então a possibilidade de surgirem problemas. Para resolver isto, foram usados apenas coleções que suportam concorrência como *CopyOnWriteArrayList* e *ConcurrentHashMap*.

Mesmo assim existem casos em que queremos alterar um objeto retornado por uma destas coleções, mesmo que muito improvável, duas threads poderiam tentar alterar este objeto ao mesmo tempo, por essa razão, todas as funções da “Database” que alteram um determinado objeto são do tipo *synchronized*.